FOODiE, Fortran Object oriented Ordinary Differential Equations integration library based on Abstract Calculus Pattern

Zaghi S.[a,1,*], Curcic M.[b,2], Rouson D.[c,3], Beekman I.[c,4]

[a] *CNR-INSEAN, Istituto Nazionale per Studi ed Esperienze di Architettura Navale, Via di Vallerano 139, Rome, Italy, 00128*
[b] *Ocean Sciences Rosenstiel School of Marine and Atmospheric Science, University of Miami, 4600 Rickenbacker Causeway Miami, FL 33149-1098 +1 305.421.4000*
[c] *Sourcery Institute 482 Michigan Ave., Berkeley, CA 94707*

Abstract

To be written.

Keywords: Ordinary Differential Equations (ODE), Partial Differential Equations (PDE), Object Oriented Programming (OOP), Abstract Calculus Pattern (ACP), Fortran,

PROGRAM SUMMARY

Manuscript Title: FOODiE, Fortran Object oriented Ordinary Differential Equations integration library based on Abstract Calculus Pattern
Authors: Zaghi, S., Curcic, M., Rouson, D., Beekman, I.
Program title: FOODiE
Journal Reference:
Catalogue identifier:
Licensing provisions: GNU General Public License (GPL) v3
Programming language: Fortran (standard 2008 or newer); developed and tested with GNU gfortran 5.2 or newer
Computer(s) for which the program has been designed: designed for shared-memory multi-cores workstations and for hybrid distributed/shared-memory supercomputers, but any computer system with a Fortran (2008+) compiler is suited
Operating system(s) for which the program has been designed: designed for POSIX architecture and tested on GNU/Linux one
RAM required to execute with typical data: bytes: $[1MB, 1GB] \times core$, simulation-dependent
Has the code been vectorised or parallelized?: the library is not aware of the parallel back-end, it providing a high-level models, but the provided tests suite shows parallel usage by means of MPI library and OpenMP paradigm
Number of processors used: tested up to 256
Supplementary material:
Keywords: ODE, PDE, OOP, ACP, Fortran
CPC Library Classification: 4.3 Differential Equations, 4.10 Interpolation, 12 Gases and Fluids
External routines/libraries used:
CPC Program Library subprograms used:
Nature of problem:
Numerical integration of (general) Ordinary Differential Equations system
Solution method:

*Corresponding author
Email addresses: stefano.zaghi@cnr.it (Zaghi S.), milan@orca.rsmas.miami.edu (Curcic M.), damian@sourceryinstitute.org (Rouson D.), izaak@izaakbeekman.com (Beekman I.)
[1] Ph. D., Aerospace Engineer, Research Scientist, Dept. of Computational Hydrodynamics at CNR-INSEAN.
[2] Ph.D. Meteorology and Physical Oceanography, Research Scientist, Dept. of Ocean Sciences Rosenstiel School of Marine and Atmospheric Science at University of Miami
[3] Ph.D. Mechanical Engineering, Founder and President Sourcery Institute and Sourcery, Inc.
[4] Graduate Research Assistant, Princeton/UMD CCROCCO LAB

Restrictions:
Unusual features:
Additional comments:
Running time:
References:

1. Introduction

1.1. Background

Initial Value Problem (IVP, or Cauchy problem) constitutes a class of mathematical models of paramount relevance, it being applied to the modelling of a wide range of problems. Briefly, an IVP is an Ordinary Differential Equations system (ODE) coupled with specified initial values of the unknown state variables, the solution of which are searched at a given time after the initial time considered.

The prototype of IVP can be expressed as:

$$\begin{aligned} U_t &= R(t, U) \\ U_0 &= U(t_0) \end{aligned} \tag{1}$$

where $U(t)$ is the vector of state variables being a function of the time-like independent variable t, $U_t = \frac{dU}{dt} = R(t, U)$ is the (vectorial) residual function and $U(t_0)$ is the (vectorial) initial conditions, namely the state variables function evaluated at the initial time t_0. In general, the residual function R is a function of the state variable U through which it is a function of time, but it can be also a direct function of time, thus in general $R = R(t, U(t))$ holds.

The problem prototype 1 is ubiquitous in the mathematical modelling of physical problems: essentially whenever an *evolutionary* phenomenon is considered the prevision (simulation) of the future solutions involves the solution of an IVP. As a matter of facts, many physical problems (fluid dynamics, chemistry, biology, evolutionary-anthropology, etc...) are described by means of an IVP.

It is worth noting that the state vector variables U and its corresponding residual function $U_t = \frac{dU}{dt} = R(t, U)$ are *problem dependent*: the number and the meaning of the state variables as well as the equations governing their evolution (that are embedded into the residual function) are different for the Navier-Stokes conservation laws with respect the Burgers one, as an example. Nevertheless, the *solvers* used for the prediction of the Navier-Stokes equations evolution (hereafter the *solution*) are the same that are used for Burgers equations time-integration. As a consequence, the solution of the IVP model prototype can be generalized, allowing the application of the same solver to many different problems, thus eliminating the necessity to re-implement the same solver for each different problem.

FOODiE library is designed for solving the generalized IVP 1, it being completely unaware of the actual problem's definition. FOODiE library provides a high-level, well-documented, simple Application Program Interface (API) for many well-known ODE integration solvers, its aims being twofold:

- provide a robust set of ODE solvers ready to be applied to a wide range of different problems;
- provide a simple framework for the rapid development of new ODE solvers.

Add citations to IVP, Cauchy, ODE references.

1.2. Related Works

To be written.

1.3. Motivations and Aims

FOODiE library must:

- be written in modern Fortran (standard 2008 or newer);
- be written by means of Object Oriented Programming (OOP) paradigm;

- be well documented;
- be Tests Driven Developed (TDD);
- be collaboratively developed;
- be free.

FOODiE, meaning Fortran Object oriented Ordinary Differential Equations integration library, has been developed with the aim to satisfy the above specifications. The present paper is its first comprehensive presentation.

The Fortran (Formula Translator) programming language is the *de facto* standard into computer science field: it strongly facilitates the effective and efficient translation of (even complex) mathematical and numerical models into an operative software without compromise on computations speed and accuracy. Moreover, its simple syntax is suitable for scientific researchers that are interested (and skilled) in the physical aspects of the numerical computations rather than computer technicians. Consequently, we develop FOODiE using Fortran language: FOODiE is written by research scientists for research scientists.

One key-point of the FOODiE development is the *problem generalization*: the problem solved must be the IVP 1 rather than any of its actual definitions. Consequently, we must rely on a *generic* implementation of the solvers. To this aim, OOP is very useful: it allows to express IVP 1 in a very concise and clear formulation that is really generic. In particular, our implementation is based on *Abstract Calculus Pattern* (ACP) concept.

The Abstract Calculus Pattern. The abstract calculus pattern provides a simple solution for the connection between the very high-level expression of IVP 1 and the eventual concrete (low-level) implementation of the ODE problem being solved. ACP essentially constitutes a *contract* based on an Abstract Data Type (ADT): we specify an ADT supporting a certain set of mathematical operators (differential and integral ones) and implement FOODiE solvers only on the basis of this ADT. FOODiE clients must formulate the ODE problem under integration defining their own concrete extensions of our ADT (implementing all the deferred operators). Such an approach defines the abstract calculus pattern: FOODiE solvers are aware of only the ADT, while FOODiE clients extend the ADT for defining the concrete ODE problem.

Is is worth noting that this ACP emancipates the solvers implementations from any low-level problem-dependent details: the ODE solvers developed with this pattern are extremely concise, clear, maintainable and less errors-prone with respect a low-level (non abstract) pattern. Moreover, the FOODiE clients can use solvers being extremely robust: as a matter of facts, FOODiE solvers are expressed in a formulation very close to the mathematical one and are tested on an extremely varying family of problems. As shown in the following, such a great flexibility does not compromise the computational efficiency.

The present paper is organized as following: in section 2 a brief description of the mathematical and numerical methods currently implemented into FOODiE is presented; in section 3 a detailed discussion on the implementation specifications is provided by means of an analytical code-listings review; in section 4 a verification analysis on the results of FOODiE applications is presented; section 5 provides an analysis of FOODiE performances under parallel frameworks scenario like the OpenMP and MPI paradigms; finally, in section 6 concluding remarks and perspectives are depicted.

Add citations to Fortran standards, OOP, TDD, free software, ACP, ADT, Rouson's book references.

2. Mathematical and Numerical Models

In many (most) circumstances, the solution of equation 1 cannot be computed in a closed, exact form (even if it exists and is unique) due to the complexity and nature of the residual functions, that is often non linear. Consequently, the problem is often solved relying on a numerical approach: the solution of system 1 at a time t^n, namely $U(t^n)$, is approximated by a subsequent time-marching approximations $U_0 = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_N \approx U(t^n)$ where the relation $u_i \rightarrow u_{i+1}$ implies a *stepping, numerical integration* from the time t^i to time t^{i+1} and N is the total number of numerical time steps necessary to evolve the initial conditions toward the searched solution $U(t^n)$. To this aim, many numerical schemes have been devised. Notably, the numerical schemes of practical usefulness must posses some necessary proprieties such as *consistency* and *stability* to ensure that the numerical approximation *converges*

to the exact solution as the numerical time step tends to zero. A detailed discussion of these details is out the scope of the present work and is omitted. Here, we briefly recall some classifications necessary to introduce the schemes implemented into the FOODiE library.

A non comprehensive classification of the most widely used schemes could distinguish between *multi-step* versus *one-step* schemes and between *explicit* versus *implicit* schemes.

Essentially, the multi-step schemes have been developed to obtain an accurate approximation of the subsequent numerical steps using the informations contained into the previously computed steps, thus this approach relates the next step approximation to a set of the previously computed steps. On the contrary, a one-step scheme evolves the solution toward the next step using only the information coming from the current time approximation. In the framework of one-step schemes family an equivalent accurate approximation can be obtained by means of a multi-stage approach as the one due to Runge-Kutta. The current version of FOODiE provides schemes belonging to both these families.

The other ODE solvers classification concerns with explicit or implicit nature of the schemes employed. Briefly, an explicit scheme computes the next step approximation using the previously computed steps at most to the current time, whereas an implicit scheme uses also the next step approximation (that is the unknown), thus it requires extra computations. The implicit approach is of practical use for *stiff* systems where the usage of explicit schemes could require an extremely small time step to evolve in a *stable* way the solution. Currently, FOODiE provides only explicit solvers.

FOODiE currently implements the following ODE schemes:

- one-step schemes:
  - explicit forward Euler scheme, it being 1^{st} order accurate;
  - explicit Runge-Kutta schemes:
    - TVD/SSP Runge-Kutta schemes:
      - 2-stages, it being 2^{nd} order accurate;
      - 3-stages, it being 3^{rd} order accurate;
      - 5-stages, it being 4^{th} order accurate;
    - low storage Runge-Kutta schemes:
      - 5-stages 2N registers schemes, it being 4^{th} order accurate;
- multi-step schemes:
  - Adams-Bashforth schemes:
    - 2-steps, it being 2^{nd} order accurate;
    - 3-steps, it being 3^{rd} order accurate;
    - 4-steps, it being 4^{th} order accurate;
  - Leapfrog schemes[5]:
    - 2-steps unfiltered, it being 2^{nd} order accurate, but mostly *unstable*;
    - 2-steps Robert-Asselin filtered, it being 1^{st} order accurate (on *amplitude error*);
    - 2-steps Robert-Asselin-Williams filtered, it being 3^{rd} order accurate (on *amplitude error*);

Add citations to all solvers reference papers.

[5]Leapfrog method is a 2-steps solver being, in general 2^{nd} order accurate, but mostly *unstable*. It is well suited for periodic/oscillatory problems, thus the error is generally categorized accordingly to the phase and amplitude errors: the unfiltered scheme provides a 2^{nd} order error on phase approximation while the amplitude error is null. However, the unfiltered scheme is mostly unstable, thus many filters have been devised. In general, the filters retain the accuracy on the phase error, while affect the amplitude one.

2.1. The explicit forward Euler scheme

The explicit forward Euler scheme for ODE integration is probably the simplest solver ever devised. Considering the system 1, the solution (approximation) of the state vector U at the time $t^{n+1} = t^n + \Delta t$ assuming to known the solution at time t^n is:

$$U\left(t^{n+1}\right) = U\left(t^n\right) + \Delta t \cdot R\left[t^n, U\left(t^n\right)\right] \tag{2}$$

where the solution at the new time step is computed by means of only the current time solution, thus this is an explicit scheme. The solution is an approximation of 1^{st} order, the local truncation error being $O(Dt^2)$. As well known, this scheme has an absolute (linear) stability locus equals to $|1 + \Delta t\lambda| \leq 1$ where λ contains the eigenvalues of the linear (or linearized) Jacobian matrix of the system.

This scheme is Total Variation Diminishing (TVD), thus satisfies the maximum principle (or the equivalent positivity preserving property).

Add citations.

2.2. The explicit TVD/SSP Runge-Kutta class of schemes

Runge-Kutta methods belong to the more general multi-stage family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1^{st} Euler scheme, but without increasing the number of time steps used, as it is done with the multi-step schemes. Essentially, the high order of accuracy is obtained by means of *intermediate values* (the stages) of the solution and its derivative are generated and used within a single time step. This commonly implies the allocation of some auxiliary memory registers for storing the intermediate stages.

Notably, the multi-stage schemes class has the attractive property to be *self-starting*: the high order accurate solution can be obtained directly from the previous one, without the necessity to compute *before* a certain number of previous steps, as it happens for the multi-step schemes. Moreover, one-step multi-stage methods are suitable for adaptively-varying time-step size (that is also possible for multi-step schemes, but at a cost of more complexity) and for discontinuous solutions, namely discontinued solutions happening at a certain time $t*$ (that in a multi-step framework can involve an overall accuracy degradation).

In general, the TVD/SSP Runge-Kutta schemes provided by FOODiE library are written by means of the following algorithm:

$$U^{n+1} = U^n + \Delta t \cdot \sum_{s=1}^{N_s} \beta_s K_s \tag{3}$$

where N_s is the number of Runge-Kutta stages used and K_s is the s^{th} stage defined as:

$$K_s = R\left(t^n + \gamma_s \Delta t, U^n + \Delta t \sum_{l=1}^{s-1} \alpha_{s,l} K_l\right) \tag{4}$$

It is worth noting that the equations 3 and 4 contain also implicit schemes. A scheme belonging to this family is operative once the coefficients α, β, γ are provided. We represent these coefficients using the Butcher's table, that for an explicit scheme where $\gamma_1 = \alpha_{1,*} = \alpha_{i,i} = 0$ has the form reported in table 1.

Table 1: Butcher's table for explicit Runge-Kutta schemes

γ_2	$\alpha_{2,1}$				
γ_3	$\alpha_{3,1}$	$\alpha_{3,2}$			
$\vdots$	$\vdots$		$\ddots$		
γ_{N_s}	$\alpha_{N_s,1}$	$\alpha_{N_s,2}$	$\cdots$	α_{N_s,N_s-1}	
	β_1	β_2	$\cdots$	β_{N_s-1}	β_{N_s}

The equations 3 and 4 show that Runge-Kutta methods do not require any additional differentiations of the ODE system for achieving high order accuracy, rather they require additional evaluations of the residual function R.

The nature of the scheme and the properties of the solutions obtained depend on the number of stages and on the value of the coefficients selected. Currently, FOODiE provides 3 Runge-Kutta schemes having TVD or Strong Stability Preserving (SSP) propriety (thus they being suitable for ODE systems involving rapidly changing non linear dynamics) the Butcher's coefficients of which are reported in tables 2, 3, 4.

Table 2: Butcher's table of 2 stages, 2^{nd} order, Runge-Kutta TVD scheme

1	1	0
	1/2	1/2

Table 3: Butcher's table of 3 stages, 3^{rd} order, Runge-Kutta SSP scheme

1	1		
1/2	1/4	1/4	
	1/6	1/6	2/3

Table 4: Butcher's table of 5 stages, 4^{th} order accurate, Runge-Kutta SSP scheme

0.39175222700392	0.39175222700392				
0.58607968896779	0.21766909633821	0.36841059262959			
0.47454236302687	0.08269208670950	0.13995850206999	0.25189177424738		
0.93501063100924	0.06796628370320	0.11503469844438	0.20703489864929	0.54497475021237	
	0.14681187618661	0.24848290924556	0.10425883036650	0.27443890091960	0.22600748319395

The absolute stability locus depends on the coefficients selected, however, as a general principle, we can assume that greater is the stages number and wider is the stability locus on equal accuracy orders.

It is worth noting that FODDiE also provides a one-stage TVD Runge-Kutta solver that reverts back to the explicit forward Euler scheme: it can be used, for example, into a Recursive Order Reduction (ROR) framework that automatically checks some properties of the solution and, in case, reduces the order of the Runge-Kutta solver until those properties are obtained.

Add citations.

2.3. The explicit low storage Runge-Kutta class of schemes

As aforementioned, standard Runge-Kutta schemes have the drawback to require N_S auxiliary memory registers to store the necessary stages data. In order to make an efficient use of the available limited computer memory, the class of low storage Runge-Kutta scheme was devised. Essentially, the standard Runge-Kutta class (under some specific conditions) can be reformulated allowing a more efficient memory management. Currently FOODiE provides a class of *2N registers* storage Runge-Kutta schemes, meaning that the storage of all stages requires only 2 registers of memory with a *word* length N (namely the length of the state vector) in contrast to the standard formulation where N_s registers of the same length N are required. This is a dramatic improvement of memory efficiency especially for schemes using a high number of stages ($N_s \geq 4$) where the memory necessary is an half with respect the original formulation. Unfortunately, not all standard Runge-Kutta schemes can be reformulated as a low storage one.

Following the Williamson's approach the standard coefficients are reformulated to the coefficients vectors A, B and C and the Runge-Kutta algorithm becomes:

$$\begin{gathered} K_1 = U\left(t^n\right) \\ K_2 = 0 \\ \left.\begin{gathered} K_2 = A_s K_2 + \Delta t \cdot R\left(t^n + C_s \Delta t, K_1\right) \\ K_1 = K_1 + B_s K_2 \end{gathered}\right\} s = 1, 2, \ldots N_s \\ U\left(t^{n+1}\right) = K_1 \end{gathered} \tag{5}$$

Table 5: Williamson's table of 5 stages, 4^{th} order, Runge-Kutta low storage scheme

A	B	C
0	$\frac{1432997174477}{9575080441755}$	0
$-\frac{567301805773}{1357537059087}$	$\frac{5161836677717}{13612068292357}$	$\frac{1432997174477}{9575080441755}$
$-\frac{2404267990393}{2016746695238}$	$\frac{1720146321549}{2090206949498}$	$\frac{2526269341429}{6820363962896}$
$-\frac{3550918686646}{2091501179385}$	$\frac{3134564353537}{4481467310338}$	$\frac{2006345519317}{3224310063776}$
$-\frac{1275806237668}{842570457699}$	$\frac{2277821191437}{14882151754819}$	$\frac{2802321613138}{2924317926251}$

Currently FOODiE provides only one 5 stages, 4^{th} order, 2N registers explicit scheme, the coefficients of which are listed in table 5.

Similarly to the TVD/SSP Runge-Kutta class, the low storage class also provides a fail-safe one-stage solver reverting back to the explicit forward Euler solver, that is useful for ROR-like frameworks.

Add citations.

2.4. The explicit Adams-Bashforth class of schemes

Adams-Bashforth methods belong to the more general (linear) multi-step family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1^{st} Euler scheme using the information coming from the solutions already computed at previous time steps. Typically only one new residual function R evaluation is required at each time step, whereas Runge-Kutta schemes require many of them.

In general, the Adams-Bashforth schemes provided by FOODiE library are written by means of the following algorithm (for only explicit schemes):

$$U\left(t^{n+N_s}\right) = U\left(t^{n+N_s-1}\right) + \Delta t \sum_{s=1}^{n+N_s} b_s \cdot R\left[t^{n+s-1}, U\left(t^{n+s-1}\right)\right] \tag{6}$$

where N_s is the number of time steps considered and b_s are the linear coefficients selected.

Currently FOODiE provides 2, 3, and 4 steps schemes having 2^{nd}, 3^{rd} and 4^{th} formal order of accuracy, respectively. The b_s coefficients are reported in table 6.

Table 6: Explicit Adams-Bashforth coefficients

N_s	b_1	b_2	b_3	b_4
2	$-\frac{1}{2}$	$\frac{3}{2}$	/	/
3	$\frac{5}{12}$	$-\frac{16}{12}$	$\frac{23}{12}$	/
4	$-\frac{9}{24}$	$\frac{37}{24}$	$-\frac{59}{24}$	$\frac{55}{24}$

Similarly to the Runge-Kutta classes, the Adams-Bashforth class also provides a fail-safe one-step solver reverting back to the explicit forward Euler solver, that is useful for ROR-like frameworks.

It is worth noting that for $N_s > 1$ the Adams-Bashforth class of solvers is not *self-starting*: the values of $U\left(t^1\right)$, $U\left(t^2\right)$, ..., $U\left(t^{N_s-1}\right)$ must be provided. To this aim, a lower order multi-step scheme or an equivalent order one-step multi-stage scheme can be used.

Add citations.

2.5. The leapfrog solver

The *leapfrog* scheme belongs to the multi-step family, it being formally a centered second order approximation in time. The leapfrog method (in its original formulation) is mostly unstable, however it is well suited for periodic-oscillatory problems providing a null error on the amplitude value and a formal second order error on the phase one, under the satisfaction of the time-step size stable limit. Commonly, the leapfrog methods are said to provide a $2\Delta t$ computational mode that can generate unphysical, unstable solutions. As consequence, the original leapfrog scheme is generally *filtered* in order to suppress these computational modes.

The unfiltered leapfrog scheme provided by FOODiE is:

$$U\left(t^{n+2}\right) = U\left(t^{n}\right) + 2\Delta t \cdot R\left[t^{n+1}, U\left(t^{n+1}\right)\right] \tag{7}$$

FOODiE provides, in a *seamless* API, also filtered leapfrog schemes. A widely used filter is due to Robert and Asselin, that suppress the computational modes at the cost of accuracy reduction resulting into a 1^{st} order error in amplitude value. A more accurate filter, able to provide a 3^{rd} order error on amplitude, is a modification of the Robert-Asselin filter due to Williams known as Robert-Asselin-Williams (RAW) filter, that filters the approximation of $U\left(t^{n+1}\right)$ and $U\left(t^{n+2}\right)$ by the following scalar coefficient:

$$\begin{gathered} U\left(t^{n+1}\right) = U\left(t^{n+1}\right) + \Delta * \alpha \\ U\left(t^{n+2}\right) = U\left(t^{n+2}\right) + \Delta * (\alpha - 1) \\ where \\ \Delta = \frac{\nu}{2}(U^{n} - 2U^{n+1} + U^{n+2}) \end{gathered} \tag{8}$$

The filter coefficients should be taken as $\nu \in [0, 1]$ and $\alpha \in [0.5, 1]$. If $\alpha = 0.5$ the filters of time t^{n+1} and t^{n+2} have the same amplitude and opposite sign thus allowing to the optimal 3^{rd} order error on amplitude. The default values of the FOODiE provided scheme are $\nu = 0.01$ $\alpha = 0.5$, but they can be customized at runtime.

Add citations.

3. Application Program Interface

In this section we review the FOODiE API providing a detailed discussion of the implementation choices.

As aforementioned, the programming language used is the Fortran 2008 standard, that is a minor revision of the previous Fortran 2003 standard. Such a new Fortran idioms provide (among other useful features) an almost complete support for OOP, in particular for ADT concept. Fortran 2003 has introduced the *abstract derived type*: it is a derived type suitable to serve as *contract* for concrete type-extensions that has not any actual implementations, rather it provides a well-defined set of type bound procedures interfaces, that in Fortran nomenclature are called *deferred* procedures. Using such an abstract definition, we can implement algorithms operating on only this abstract type and on *all its concrete extensions*. This is the key feature of FOODiE library: all the above described ODE solvers are implemented on the knowledge of *only one abstract type*, allowing an implementation-style based on a very high-level syntax. In the meanwhile, client codes must implement their own IVPs extending only one simple abstract type.

In the subsection 3.1 a review of the FOODiE main ADT, the *integrand* type, is provided, while subsections 3.2, 3.3, 3.4, 3.5 and 3.6 cover the API of the currently implemented solvers.

It is worth noting that all FOODiE public *entities* (ADT and solvers) must be accessed by the FOODiE module, see listing 1 for an example on how access to all public FOODiE entities.

```
use foodie, only: integrand, &
                  adams_bashforth_integrator, &
                  euler_explicit_integrator, &
                  leapfrog_integrator, &
                  ls_runge_kutta_integrator, &
                  tvd_runge_kutta_integrator
! or simply
use foodie
```

Listing 1: usage example importing all public entities of FOODiE main module

3.1. The main FOODiE Abstract Data Type: the integrand type

The implemented ACP is based on one main ADT, the *integrand* type, the definition of which is shown in listing 2.

```
type, abstract :: integrand
  !< Abstract type for building FOODiE ODE integrators.
  contains
    ! public deferred procedures that concrete integrand-field must implement
    procedure(time_derivative),        pass(self), deferred, public :: t
    procedure(update_previous_steps), pass(self), deferred, public :: update_previous_steps
    procedure(previous_step),          pass(self), deferred, public :: previous_step
    ! operators
    procedure(symmetric_operator),   pass(lhs), deferred, public :: integrand_multiply_integrand
    procedure(integrand_op_real),    pass(lhs), deferred, public :: integrand_multiply_real
    procedure(real_op_integrand),    pass(rhs), deferred, public :: real_multiply_integrand
    procedure(symmetric_operator),   pass(lhs), deferred, public :: add
    procedure(symmetric_operator),   pass(lhs), deferred, public :: sub
    procedure(assignment_integrand), pass(lhs), deferred, public :: assign_integrand
    ! operators overloading
    generic, public :: operator(+) => add
    generic, public :: operator(-) => sub
    generic, public :: operator(*) => integrand_multiply_integrand, &
                                      real_multiply_integrand, &
                                      integrand_multiply_real
    generic, public :: assignment(=) => assign_integrand
endtype integrand
```

Listing 2: integrand type definition

The *integrand* type does not implement any actual integrand field, it being and abstract type. It only specifies which deferred procedures are necessary for implementing an actual concrete integrand type that can use a FOODiE solver.

As shown in listing 2, the number of the deferred type bound procedures that clients must implement into their own concrete extension of the *integrand* ADT is very limited: essentially, there are 3 ODE-specific procedures plus some operators definition constituted by symmetric operators between 2 integrand objects, asymmetric operators between integrand and real numbers (and viceversa) and an assignment statement for the creation of new integrand objects. These procedures are analyzed in the following paragraphs.

3.1.1. Time derivative procedure, the residual function

The abstract interface of the time derivative procedure *t* is shown in listing 3.

```
function time_derivative(self, n, t) result(dState_dt)
import :: integrand, R_P, I_P
class(integrand), intent(IN) :: self !< Integrand field.
integer(I_P), optional, intent(IN) :: n !< Time level.
real(R_P), optional, intent(IN) :: t !< Time.
class(integrand), allocatable :: dState_dt !< Result of the time derivative function of integrand field.
endfunction time_derivative
```

Listing 3: time derivative procedure interface

This procedure-function takes three arguments, the first passed as a *type bounded* argument, while the latter are optionals, and it returns an integrand object. The passed dummy argument, *self*, is a polymorphic argument that could be any extensions of the *integrand* ADT. The optional argument *n* indicates the *time level* at which the residual function must be evaluated: this argument is necessary for multi-step (often referred as multi-level) solvers such as the Adams-Bashforth or leapfrog ones, but it can be omitted for other classes of solvers such the one-step multi-stages family. The optional argument *t* is the *time* at which the residual function must be computed: it can be omitted in the case the residual function does not depend directly on time.

Commonly, into the concrete implementation of this deferred abstract procedure clients embed the actual ODE equations being solved. As an example, for the Burgers equation, that is a Partial Differential Equations (PDE) system involving also a boundary value problem, this procedure embeds the spatial operator that convert the PDE to a system of algebraic ODE. As a consequence, the eventual concrete implementation of this procedure can be very complex and errors-prone. Nevertheless, the FOODiE solvers are implemented only on the above abstract interface, thus emancipating the solvers implementation from any concrete complexity.

Add citations to Burgers, Adams-Bashfort, leapfrog references.

3.1.2. Update previous time steps procedure

The abstract interface of the *update previous time steps* procedure is shown in listing 4.

```
subroutine update_previous_steps(self, filter, weights)
import :: integrand, R_P
class(integrand), intent(INOUT) :: self !< Integrand field.
class(integrand), optional, intent(IN) :: filter !< Filter field displacement.
real(R_P), optional, intent(IN) :: weights(:) !< Weights for filtering the steps.
endsubroutine update_previous_steps
```

Listing 4: update previous time steps procedure interface

This procedure is a necessary complement for using multi-step solvers. As its name suggests, it updates the previous time steps solutions (that must be stored on the care of the clients code). It takes three arguments, the first passed as a *type bounded* argument, while the latter are optionals, and it returns nothing it being a subroutine directly modifying the passed argument. The passed dummy argument, *self*, is a polymorphic argument that could be any extensions of the *integrand* ADT. The optional argument *filter* indicates a *displacement* for filtering the field while the last optional argument *weights* specifies the weights that must applied to the displacement for each time steps filtered.

Let us consider the multi-step scheme expressed by the equation 6. Once the solution at time t^{n+N_s} has been computed the previous $N_s - 1$ steps must be updated, e.g. an unfiltered update can read as:

$$U\left(t^{n+s}\right) = U\left(t^{n+s+1}\right) \quad s = 1, .., N_s - 1 \tag{9}$$

and a filtered update can read as:

$$U\left(t^{n+s}\right) = \Delta * U\left(t^{n+s+1}\right)\omega_s \quad s = 1, .., N_s - 1 \tag{10}$$

where Δ is the filter displacement and ω_s the weight of the step $s - th$. This interface implies that the previous time steps must be embedded into the client *integrand* extension. This procedure is automatically called by each multi-step solver after the new step t^{n+N_s} has been computed, thus alleviating client codes from its explicit call.

In the case the client codes will use only one-step solvers this procedure can be defined as simple *empty* subroutine as shown in listing 5.

```
subroutine update_previous_steps(self, filter, weights)
class(my_integrand), intent(INOUT) :: self    !< Integrand field.
class(integrand), optional, intent(IN)  :: filter  !< Filter field displacement.
real(R_P), optional, intent(IN)  :: weights(:) !< Weights for filtering the steps.
! do nothing
return
endsubroutine update_previous_steps
```

Listing 5: empty implementation of the update previous time steps procedure

where *my_integrand* is a concrete (valid) extension of *integrand* ADT.

3.1.3. Previous step procedure

The abstract interface of the *previous step* procedure is shown in listing 6.

```
function previous_step(self, n) result(previous)
import :: integrand, I_P
class(integrand), intent(IN)  :: self     !< Integrand field.
integer(I_P), intent(IN)  :: n        !< Time level.
class(integrand), allocatable :: previous !< Selected previous time integrand field.
endfunction previous_step
```

Listing 6: previous step procedure interface

This procedure is a necessary complement for using multi-step solvers. It returns the integrand solution at the specified previous time step (level). It takes two arguments and it returns an integrand object. The first argument passed as a *type bounded* argument is the current instance of the integrand field object. The second argument, named *n*, indicates the *time level* at which the solution must be returned.

This procedure is used in multi-step methods like the leapfrog where there is an explicit dependency on also the solution at previous time steps.

For the case in which multi-step solvers are not used, this procedure can have a concrete implementation that does nothing as shown in listing 7.

```
function previous_step(self, n) result(previous)
class(my_integrand), intent(IN)  :: self     !< Integrand field.
integer(I_P), intent(IN)  :: n        !< Time level.
class(integrand), allocatable    :: previous !< Selected previous time integrand field.
! do nothing
return
endfunction previous_step
```

Listing 7: empty implementation of the previous step procedure

where *my_integrand* is a concrete (valid) extension of *integrand* ADT. Note that, in this case, the returned result is not allocated.

3.1.4. Symmetric operators procedures

The abstract interface of *symmetric* procedures is shown in listing 8.

```
function symmetric_operator(lhs, rhs) result(operator_result)
import :: integrand
class(integrand), intent(IN)  :: lhs             !< Left hand side.
class(integrand), intent(IN)  :: rhs             !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction symmetric_operator
```

Listing 8: symmetric operator procedure interface

This interface defines a class of procedures operating on 2 *integrand* objects, namely it is used for the definition of the operators *multiplication*, *summation* and *subtraction* of integrand objects. These operators are used into the above described ODE solvers, for example see equations 2, 3, 6 or 7. The implementation details of such a procedures class are strictly dependent on the concrete extension of the integrand type. From the FOODiE solvers point of view, we need to known only that first argument passed as bounded one, the left-hand-side of the operator, and the second argument, the right-hand-side of the operator, are two integrand object and the returned object is still an integrand one.

3.1.5. Integrand/real and real/integrand operators procedures

The abstract interfaces of *Integrand/real and real/integrand operators* procedures are shown in listing 9.

```
function integrand_op_real(lhs, rhs) result(operator_result)
import :: integrand, R_P
class(integrand), intent(IN)  :: lhs             !< Left hand side.
real(R_P),        intent(IN)  :: rhs             !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction integrand_op_real

function real_op_integrand(lhs, rhs) result(operator_result)
import :: integrand, R_P
real(R_P),        intent(IN)  :: lhs             !< Left hand side.
class(integrand), intent(IN)  :: rhs             !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction real_op_integrand
```

Listing 9: Integrand/real and real/integrand operators procedure interfaces

These two interfaces are necessary in order to complete the *algebra* operating on the integrand object class, allowing the multiplication of an integrand object for a real number, circumstance that happens in all solvers, see equations 2, 3, 6 or 7. The implementation details of these procedures are strictly dependent on the concrete extension of the integrand type. From the FOODiE solvers point of view, we need to known only that first argument passed as bounded one, the left-hand-side of the operator, and the second argument, the right-hand-side of the operator, are an integrand object and real number of viceversa and the returned object is still an integrand one.

3.1.6. Integrand assignment procedure

The abstract interface of *integrand assignment* procedure is shown in listing 10.

```
subroutine assignment_integrand(lhs, rhs)
import :: integrand
class(integrand), intent(INOUT) :: lhs !< Left hand side.
class(integrand), intent(IN)    :: rhs !< Right hand side.
endsubroutine assignment_integrand
```

Listing 10: integrand assignment procedure interface

The assignment statement is necessary in order to complete the *algebra* operating on the integrand object class, allowing the assignment of an integrand object by another one, circumstance that happens in all solvers, see equations 2, 3, 6 or 7. The implementation details of this assignment is strictly dependent on the concrete extension of the integrand type. From the FOODiE solvers point of view, we need to known only that first argument passed as bounded one, the left-hand-side of the assignment, and the second argument, the right-hand-side of the assignment, are two integrand objects.

3.2. The explicit forward Euler solver

The explicit forward Euler solver is exposed (by the FOODiE main module that must imported, see listing 1) as a single derived type (that is a standard convention for all FOODiE solvers) named *euler_explicit_integrator*. It provides the type bound procedure (also referred as *method*) *integrate* for integrating in time an *integrand* object, or any of its polymorphic concrete extensions. Consequently, for using such a solver it must be previously defined as an instance of the exposed FOODiE integrator type, see listing 11.

```
use FOODiE, only: euler_explicit_integrator
type(euler_explicit_integrator) :: integrator
```

Listing 11: definition of an explicit forward Euler integrator

Once an integrator of this type has been instantiated, it can be directly used without any initialization, for example see listing 12.

```
type(my_integrand) :: my_field
call integrator%integrate(U=my_field, Dt=0.1)
```

Listing 12: example of usage of an explicit forward Euler integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT.

The complete implementation of the *integrate* method of the explicit forward Euler solver is reported in listing 13.

```
subroutine integrate(U, Dt, t)
class(integrand),    intent(INOUT) :: U  !< Field to be integrated.
real(R_P),           intent(IN)    :: Dt !< Time step.
real(R_P), optional, intent(IN)    :: t  !< Time.
U = U + U%t(t=t) * Dt
return
endsubroutine integrate
```

Listing 13: implementation of the *integrate* method of Euler solver

This method takes three arguments, the first argument is an integrand class, it being the integrand field that must integrated one-step-over in time, the second is the time step used and the third, that is optional, is the current time value that is passed to the residual function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

As shown from listing 13 the *integrand* ADT has allowed a very high-level implementation of the Euler scheme. For example let us compare the implementation instructions with the Euler's scheme equation 2:

$$U\left(t^{n+1}\right) = U\left(t^n\right) + \Delta t \cdot R\left[t^n, U\left(t^n\right)\right] \rightarrow U = U + U\%t(t = t) * Dt$$

As shown, the Fortran implementation is almost equivalent to the rigorous mathematical formulation. This aspect directly implies that the implementation of Euler's scheme (as of all other schemes) is very clear, concise and less-errors-prone than an *hard-coded* implementation where the Euler's solver must be implemented for each specific definition of the integrand U.

3.3. The explicit TVD/SSP Runge-Kutta class of solvers

The TVD/SSP Runge-Kutta class of solvers is exposed as a single derived type named *tvd_runge_kutta_integrator*. This type provides three methods:

- *init*: initialize the integrator among the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time.

As common for FOODiE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODiE integrator type, see listing 14.

```
use FOODiE, only: tvd_runge_kutta_integrator
type(tvd_runge_kutta_integrator) :: integrator
```

Listing 14: definition of an explicit TVD/SSP Runge-Kutta integrator

Once an integrator of this type has been instantiated, it must be initialized before used, for example see listing 15.

```
call integrator%init(stages=3)
```

Listing 15: example of initialization of an explicit TVD/SSP Runge-Kutta integrator

In the listing 15 a 3-stages solver has been initialized. As a matter of facts, from the equations 3 and 4 a solver belonging to this class is completely defined once the number of stages adopted has been chosen. The complete definition of the *tvd_runge_kutta_integrator* type is reported into listing 16. As shown, the Butcher's coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

```
type :: tvd_runge_kutta_integrator
  integer(I_P)           :: stages=0   ! Number of stages.
  real(R_P), allocatable :: alph(:,:) ! alpha Butcher's coefficients.
  real(R_P), allocatable :: beta(:)    ! beta Butcher's coefficients.
  real(R_P), allocatable :: gamm(:)    ! gamma Butcher's coefficients.
  contains
    procedure, pass(self), public :: destroy   ! Destroy the integrator.
    procedure, pass(self), public :: init      ! Initialize (create) the integrator.
    procedure, pass(self), public :: integrate ! Integrate integrand field.
    final                         :: finalize  ! Finalize object.
endtype tvd_runge_kutta_integrator
```

Listing 16: definition of *tvd_runge_kutta_integrator* type

After the solver has been initialized it can used for integrating an integrand field, as shown in listing 17.

```
type(my_integrand) :: my_field
type(my_integrand) :: my_stages(1:3)
call integrator%integrate(U=my_field, stage=my_stage, Dt=0.1)
```

Listing 17: example of usage of a TVD/SSP Runge-Kutta integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT. Listing 17 shows that the memory registers necessary for storing the Runge-Kutta stages must be supplied by the client code.

The complete implementation of the *integrate* method of the explicit TVD/SSP Runge-Kutta class of solvers is reported in listing 18.

```
subroutine integrate(self, U, stage, Dt, t)
class(tvd_runge_kutta_integrator), intent(IN)    :: self     !< Actual RK integrator.
class(integrand),                  intent(INOUT) :: U        !< Field to be integrated.
class(integrand),                  intent(INOUT) :: stage(1:) !< Runge-Kutta stages [1:stages].
real(R_P),                         intent(IN)    :: Dt       !< Time step.
real(R_P),                         intent(IN)    :: t        !< Time.
integer(I_P)                                     :: s        !< First stages counter.
integer(I_P)                                     :: ss       !< Second stages counter.
select type(stage)
class is(integrand)
  do s=1, self%stages
    stage(s) = U
    do ss=1, s - 1
      stage(s) = stage(s) + stage(ss) * (Dt * self%alph(s, ss))
    enddo
    stage(s) = stage(s)%t(t=t + self%gamm(s) * Dt)
  enddo
  do s=1, self%stages
    U = U +  stage(s) * (Dt * self%beta(s))
  enddo
endselect
return
endsubroutine integrate
```

Listing 18: implementation of the *integrate* method of TVD/SSP Runge-Kutta class

This method takes five arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must integrated one-step-over in time, the third is the stages array for storing the stages computations, the fourth is the time step used and the fifth, that is optional, is the current time value that is passed to the residual function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the stages memory registers, namely the array *stage*, must be passed as argument because it is defined as a *not-passed* polymorphic argument, thus we are not allowed to define it as an automatic array of the *integrate* method.

As it happens for the Euler's scheme, the *integrand* ADT greatly simplifies the Runge-Kutta algorithm implementation. For example let us compare the implementation instructions with the Runge-Kutta equations 3 and 4:

$$\begin{matrix} K_s = R\left(t^n + \gamma_s \Delta t, U^n + \Delta t \sum_{l=1}^{s-1} \alpha_{s,l} K_l\right) \\ U^{n+1} = U^n + \Delta t \cdot \sum_{s=1}^{N_s} \beta_s K_s \end{matrix} \rightarrow \begin{matrix} stage(s) = stage(s) + stage(ss) * (Dt * self\%alph(s, ss)) \\ U = U + stage(s) * (Dt * self\%beta(s)) \end{matrix}$$

where the syntax "(s)" and "(ss)" imply the summation operation. As for the Euler's scheme, the Fortran implementation is almost equivalent to the rigorous mathematical formulation.

3.4. The explicit low storage Runge-Kutta class of solvers

To be written.

3.5. The explicit Adams-Bashforth class of solvers

To be written.

3.6. The leapfrog solver

To be written.

4. Tests and Examples

To be written.

4.1. Oscillation equations test

To be completed.

$$\begin{aligned} U_t &= R(U) \\ U = \begin{bmatrix} x \\ y \end{bmatrix} \quad R(U) &= \begin{bmatrix} -fy \\ fx \end{bmatrix} \end{aligned} \tag{11}$$

where the frequency is chosen as $f = 10^4$. The ODE system 11 is completed by the following initial conditions:

$$\begin{aligned} x(t_0) &= 0 \\ y(t_0) &= 1 \end{aligned} \tag{12}$$

where $t_0 = 0$ is the initial time considered. The exact solution is:

$$\begin{aligned} x(t_0 + \Delta t) &= X_0 cos(f\Delta t) - y_0 sin(f\Delta t) \\ y(t_0 + \Delta t) &= X_0 sin(f\Delta t) + y_0 cos(f\Delta t) \end{aligned} \tag{13}$$

where Δt is an arbitrary time step.

4.1.1. Errors Analysis

For the analysis of the accuracy of each solvers implemented into FOODiE library, we have integrated the Oscillation equations 11 with different, decreasing time steps in the range $[5000, 2500, 1250, 625, 320, 100]$.

The error is estimated by the L2 norm of the difference between the exact (U_e) and the numerical ($U_{\Delta t}$) solutions for each time step:

$$\varepsilon(\Delta t) = ||U_e - U_{\Delta t}||_2 = \sqrt{\sum_{s=1}^{N_s} (U_e(t_0 + s * \Delta t) - U_{\Delta t}(t_0 + s * \Delta t))^2} \tag{14}$$

Using two pairs of subsequent-decreasing time steps solution is possible to estimate the order of accuracy of the solver employed computing the *observed order* of accuracy:

$$p = \frac{log10\left(\frac{\varepsilon(\Delta t_1)}{\varepsilon(\Delta t_2)}\right)}{log10\left(\frac{\Delta t_1}{\Delta t_2}\right)} \tag{15}$$

where $\frac{\Delta t_1}{\Delta t_2} > 1$.

Table 7 summarizes the errors analysis.

Adams-Bashforth.

Leapfrog.

Low Storage Runge-Kutta.

TVD/SSP Runge-Kutta.

Table 7: Oscillation test errors analysis

Solver	Time Step	Error X	Error Y	Order X	Order Y
Adams-Bashforth 2 steps	5000.0	0.109E+04	0.112E+04	/	/
	2500.0	0.246E+02	0.243E+02	5.47	5.53
	1250.0	0.783E+01	0.789E+01	1.65	1.62
	625.0	0.268E+01	0.272E+01	1.55	1.54
	320.0	0.976E+00	0.990E+00	1.51	1.51
	100.0	0.171E+00	0.173E+00	1.50	1.50
Adams-Bashforth 3 steps	5000.0	0.816E+01	0.809E+01	/	/
	2500.0	0.266E+01	0.264E+01	1.62	1.62
	1250.0	0.120E+01	0.120E+01	1.15	1.14
	625.0	0.545E+00	0.543E+00	1.14	1.14
	320.0	0.223E+00	0.222E+00	1.34	1.34
	100.0	0.418E-01	0.416E-01	1.44	1.44
leapfrog 2 steps	5000.0	0.151E+02	0.152E+02	/	/
	2500.0	0.500E+01	0.504E+01	1.60	1.59
	1250.0	0.190E+01	0.193E+01	1.40	1.39
	625.0	0.181E+01	0.181E+01	0.07	0.09
	320.0	0.157E+01	0.157E+01	0.21	0.21
	100.0	0.100E+01	0.100E+01	0.39	0.39
low storage Runge-Kutta 5 stages	5000.0	0.120E+00	0.122E+00	/	/
	2500.0	0.106E-01	0.107E-01	3.51	3.51
	1250.0	0.935E-03	0.947E-03	3.50	3.50
	625.0	0.826E-04	0.836E-04	3.50	3.50
	320.0	0.793E-05	0.803E-05	3.50	3.50
	100.0	0.135E-06	0.137E-06	3.50	3.50
TVD/SSP Runge-Kutta 2 stages	5000.0	0.316E+02	0.319E+02	/	/
	2500.0	0.892E+01	0.894E+01	1.83	1.84
	1250.0	0.301E+01	0.305E+01	1.57	1.55
	625.0	0.106E+01	0.107E+01	1.51	1.51
	320.0	0.387E+00	0.392E+00	1.50	1.50
	100.0	0.676E-01	0.685E-01	1.50	1.50
TVD/SSP Runge-Kutta 3 stages	5000.0	0.255E+01	0.252E+01	/	/
	2500.0	0.523E+00	0.516E+00	2.28	2.29
	1250.0	0.944E-01	0.931E-01	2.47	2.47
	625.0	0.167E-01	0.165E-01	2.50	2.50
	320.0	0.314E-02	0.310E-02	2.50	2.50
	100.0	0.171E-03	0.169E-03	2.50	2.50
TVD/SSP Runge-Kutta 5 stages	5000.0	0.139E+00	0.141E+00	/	/
	2500.0	0.122E-01	0.124E-01	3.50	3.50
	1250.0	0.108E-02	0.110E-02	3.50	3.50
	625.0	0.956E-04	0.969E-04	3.50	3.50
	320.0	0.937E-05	0.949E-05	3.47	3.47
	100.0	0.512E-06	0.519E-06	2.50	2.50

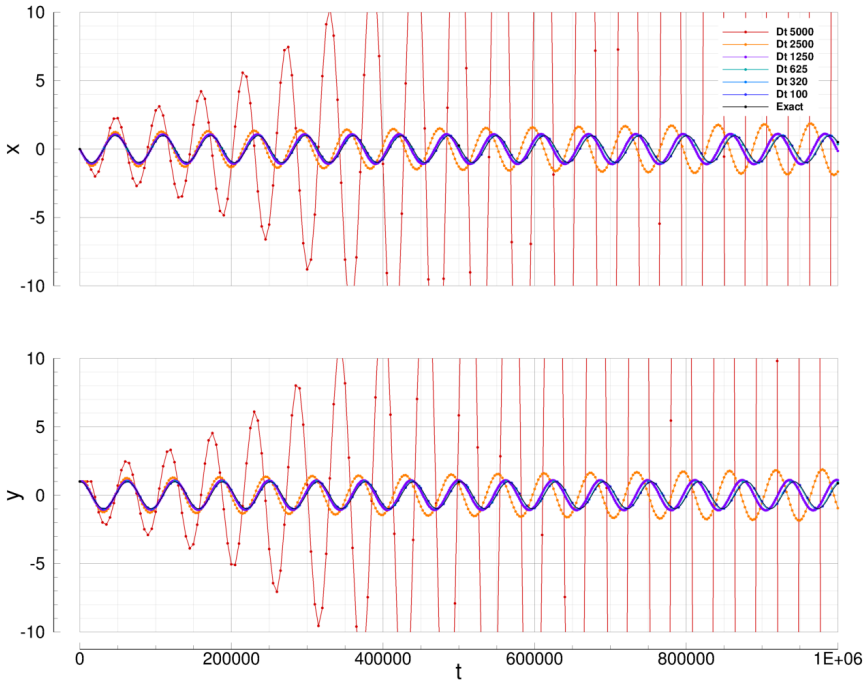

Figure 1: Oscillation equations solutions computed by means of Adams-Bashforth 2 steps solver

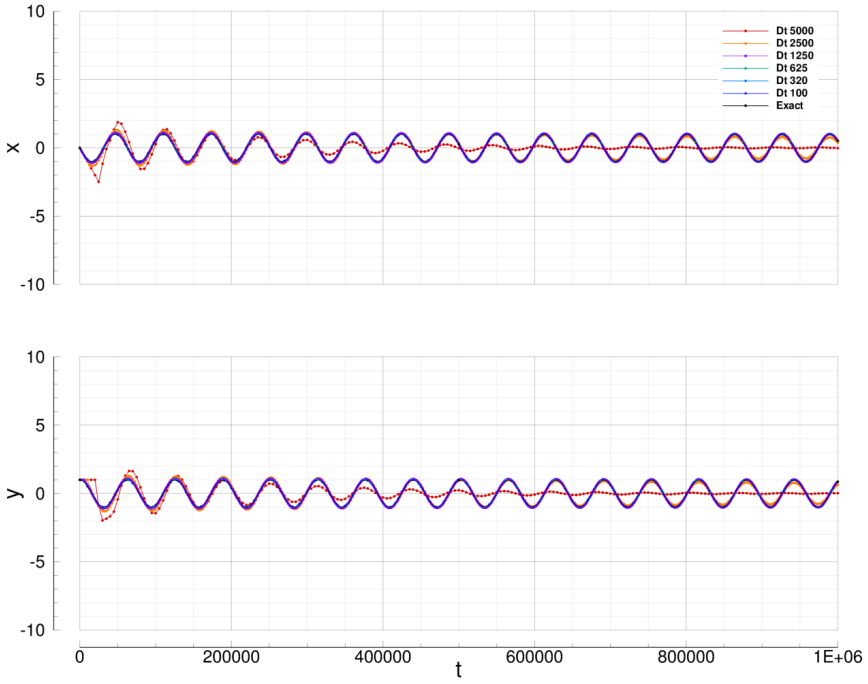

Figure 2: Oscillation equations solutions computed by means of Adams-Bashforth 3 steps solver

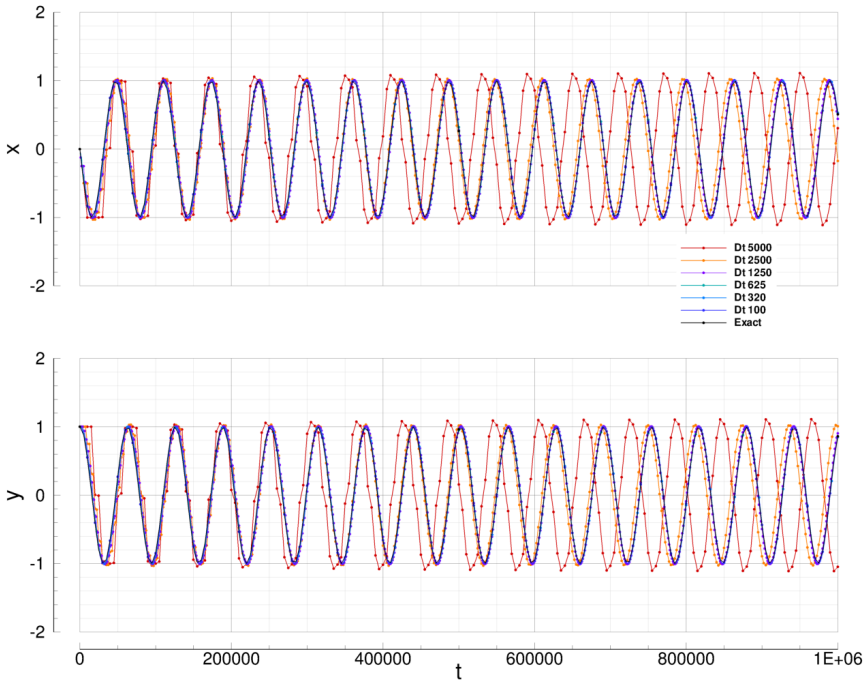

Figure 3: Oscillation equations solutions computed by means of leapfrog 2 steps solver

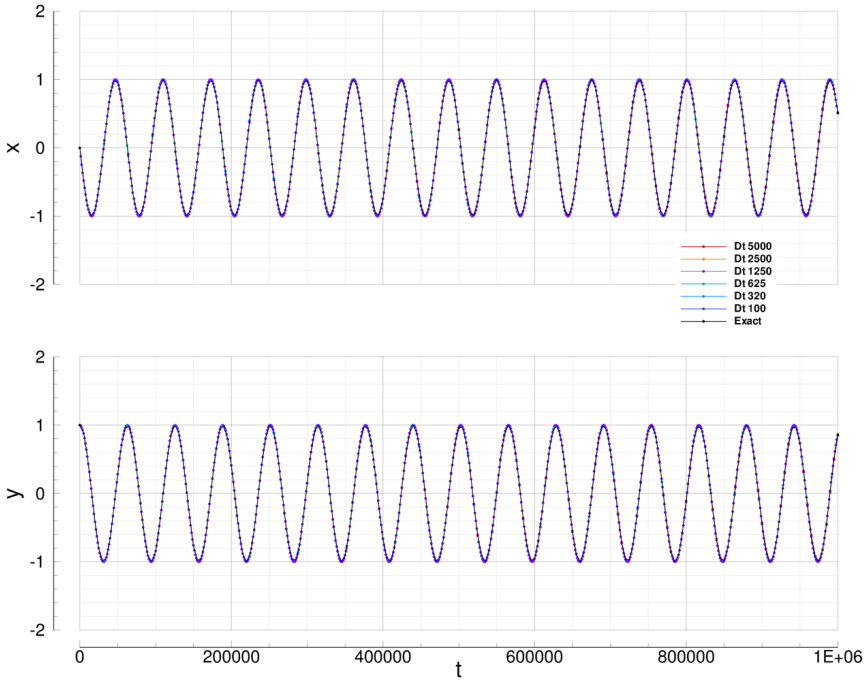

Figure 4: Oscillation equations solutions computed by means of low storage Runge-Kutta 5 stages solver

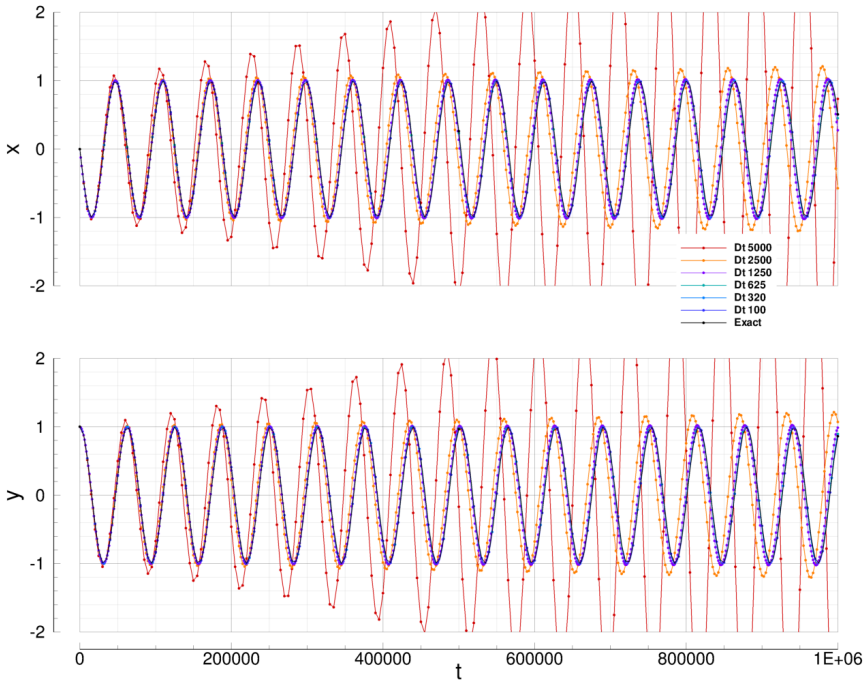

Figure 5: Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta 2 stages solver

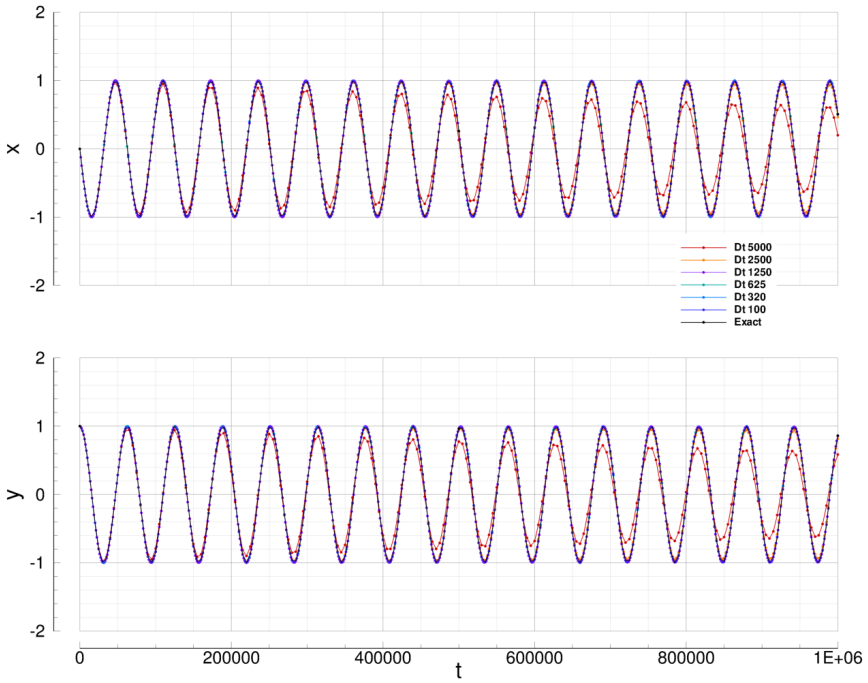

Figure 6: Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta 3 stages solver

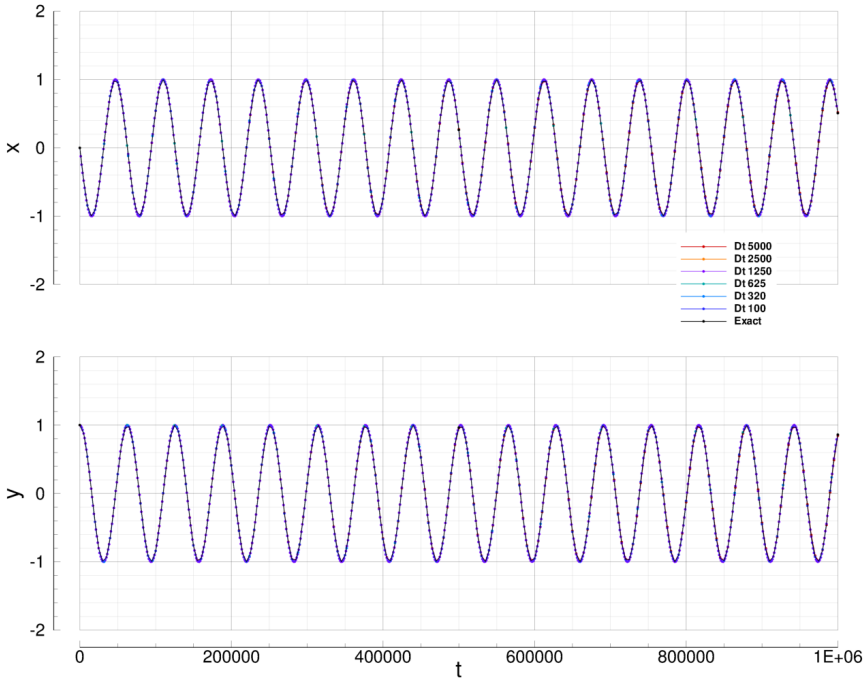

Figure 7: Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta 5 stages solver

5. Benchmarks on parallel frameworks

To be written.

5.1. OpenMP benchmark

To be written.

5.2. MPI benchmark

To be written.

6. Concluding Remarks and Perspectives

To be written.

References